Prefix-free Graphs and Suffix Array Construction in Sublinear Space

Andrej Baláž[1,*], Alessia Petescia[1]

[1] *Department of Applied Informatics, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia*

Abstract

A recent paradigm shift in bioinformatics from a single reference genome to a pangenome brought with it several graph structures. These graph structures must implement operations, such as efficient construction from multiple genomes and read mapping. Read mapping is a well-studied problem in sequential data, and, together with data structures such as suffix array and Burrows-Wheeler transform, allows for efficient computation. Attempts to achieve comparatively high performance on graphs bring many complications since the common data structures on strings are not easily obtainable for graphs. In this work, we introduce prefix-free graphs, a novel pangenomic data structure; we show how to construct them and how to use them to obtain well-known data structures from stringology in sublinear space, allowing for many efficient operations on pangenomes.

Keywords

computational pangenomics, graph pangenome, suffix array

1. Introduction

The term pangenome was first used by Tettelin et al. [1] in 2005 while studying variations in the population of Streptococcus agalactiae. Since then, pangenomes have found applications in the study of many organisms, from viruses [2] through microbes [3] and plants [4] to humans [5]. As per the definition by The Computational Pan-Genomics Consortium [6], a pangenome is any set of genomic sequences meant to be analyzed jointly. Nevertheless, in practice, most pangenomes consist of genomic sequences of highly related organisms and therefore are highly repetitive. Representation of this repetitive dataset by simple text is often inefficient and limits scaling in terms of algorithmic time and space complexity. These limitations lead to the idea of representing pangenomes as graphs, where similar genomic regions are unified into nodes, and these nodes are connected to paths representing the original genomic sequences.

Several approaches to pangenomic graph construction exist, such as variation graphs [7, 8, 9], cactus graphs [10, 11], and Wheeler graphs [12, 13]. Most of these approaches require an initial local alignment of similar regions or a multiple sequence alignment, which makes them computationally expensive. Here we present a new class of graphs, prefix-free graphs, which are orders of magnitude faster to construct. Furthermore, we explore the connection between prefix-free graphs and suffix arrays.

A suffix array is a data structure from the stringology field with a massive impact on designing many efficient algorithms on strings. Particularly in bioinformatics, it is responsible for the design of such data structures as the Burrows-Wheeler transform [14] and FM-index [15], which in turn allowed for efficient mapping of reads to the reference and several other fundamental bioinformatics operations. These fundamental operations are well-studied on sequential data, but the recent paradigm shift of moving from a single reference genome to a graph pangenome made it even more complicated to apply the acquired knowledge from the stringology field to biological sequences.

Thanks to the link to suffix arrays, prefix-free graphs have great potential to draw from this extensive knowledge. Using the suffix array from a prefix-free graph, we can obtain several stringological data structures which are not easily obtainable for graphs. This feature of prefix-free graphs was implicitly demonstrated in several articles [16, 17, 13, 18], where the authors used similar techniques as presented here. We think that explicitly defining and framing the prefix-free graph as a standalone pangenomic data structure can bring several benefits:

- reduction in the complexity of the presentation of several space-efficient algorithms

ITAT'23: Information technologies – Applications and Theory, September 22–26, 2023, Tatranské Matliare, Slovakia

*Corresponding author

EMAIL: andrejbalaz001@gmail.com (A. Baláž)

- support of theoretical research by clearly delimiting the relevant terms
- improved focus on the optimization of algorithms related to prefix-free graphs
- enabling bringing prefix-free graphs closer to the biological data

In this work, we define prefix-free graphs and show how they can be constructed from the pangenome in its textual representation. Furthermore, we show in detail how prefix-free graphs can be used to generate the suffix array of a pangenome in sublinear space and linear time. Finally, we implement the presented algorithms as two binaries for easy construction of prefix-free graphs from a set of sequences in FASTA format and from a pangenomic graph in GFA format. Furthermore, we implement the rust library for working with prefix-free graphs. This library contains an iterator, which can be directly used to generate the suffix array in sublinear space.

2. Prefix-free graphs

The idea of prefix-free graphs is inspired by a technique used in the tool rsync named Context-Triggered Piecewise Hashing (CTPH) [19]. CTPH uses a rolling hash to partition a string into substrings such that long repeated substrings are partitioned the same way. These substrings are then hashed with a traditional hash function and stored as a string signature. The signatures of several files are then compared to determine changes.

In prefix-free graphs, we partition each sequence of a given pangenome into *segment*s. These segments form nodes of the prefix-free graph, and their adjacencies in the original sequences constitute edges. The sequences are represented as *path*s in the graph.

The segments have two essential characteristics making them a good choice for nodes of a pangenomic graph. Similarly to CTPH, long repeated sequences will be partitioned the same way. Furthermore, no segment is a prefix of another, making a set of segments prefix-free. The second characteristic is crucial for connecting prefix-free graphs and suffix arrays, as will be presented in the next section.

To create a prefix-free graph from a given pangenome, we define a set of trigger words T, where each trigger word is a string of length k. For this set T, we build an Aho-Corasick automaton [20]. Then, for each sequence in the pangenome, we append k sentinel characters and iterate over such modified sequence, searching for matches with set

```
CTCTTCTGGGTAC
CTCTTCTGGGTACTATAGAAC
```

Figure 1: A sketch of a visual proof that the trigger words induce prefix-free segments. Suppose a segment is a prefix of another segment. By construction, it ends with a trigger word. This trigger word would be in the middle of another segment and would break it into two smaller segments.

T using the automaton. Each time we encounter a trigger word, we recognize a new segment from the start of the previous trigger word to the end of the current trigger word. If the segment's sequence was not yet observed during the scan, we add it to the set of segments and assign a unique ID. Each time we append the corresponding ID to the path representing the original sequence.

Two special cases happen during the sequence scan, one at the beginning, when no previous trigger word was encountered, and another at the end, when the last k characters are sentinels. These are addressed by simply starting the first segment at the start of a sequence and ending the last at the sequence end.

Notably, the adjacent segments overlap by exactly k characters. Furthermore, trigger words occur only at the beginnings or ends of segments because any occurrence of a trigger word in the middle of a segment would break it into two. This feature and the choice of sentinels outside the sequence's alphabet guarantee that the set of segments is prefix-free. A sketch of the proof is shown in Figure 1.

After the previous steps, the set of segments and the list of paths already represent a prefix-free graph. However, to simplify the usage of prefix-free graphs, we recommend normalizing them. During the normalization, we sort segments lexicographically and change their IDs to correspond to the lexicographical ranks. This relabeling is then also propagated to paths accordingly. All construction steps can be performed in space proportional to the sum of segment lengths and the sum of path lengths, which is expected to be significantly smaller than the length of the pangenome.

To illustrate the entire procedure, consider a set of sequences {`CACGTACT`, `CACACT`, `CACGACT`} and a set of trigger words $T = \{$`AC`, `CG`$\}$. After the partitioning, we obtain a set of segments with IDs {`0:CAC`, `1:ACG`, `2:CGTAC`, `3:ACT..`, `4:ACAC`, `5:CGAC`} and a list of paths `[[0,1,2,3], [0,4,3], [0,1,5,3]]`. After the normalization, we get a prefix-free graph which can be directly represented in GFA format as shown in Figure 2.

```
H VN:Z:1.1
S 0 ACAC
S 1 ACG
S 2 ACT..
S 3 CAC
S 4 CGAC
S 5 CGTAC
P 0 3+,1+,5+,2+  *
P 1 3+,0+,2+     *
P 2 3+,1+,4+,2+  *
```

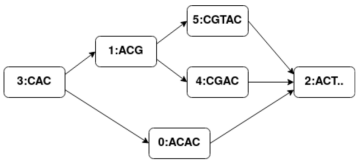

Figure 2: Prefix-free graph of the running example after normalization and its representation in GFA format. Link lines omitted for brevity.

From this representation, original sequences of a pangenome can be reconstructed by expanding the segment IDs in a particular path, ignoring the last k characters of each segment.

3. Suffix array construction

A suffix array is a permutation of string positions which lexicographically sorts the suffixes of the string starting at that position. For a set of strings, we consider a concatenation of the strings as shown in Figure 4. In the following text, we will refer to this concatenation as the pangenome.

The suffix array is an influential data structure with many applications in efficient string algorithms solving problems such as exact pattern matching, repeat finding, maximum exact match (MEM) finding, document retrieval and many more. There exist several algorithms for suffix array construction in linear time [21, 22, 23] with several practical implementations [24, 25]. Despite their linear time complexity, these algorithms become bottlenecks in some applications because of their linear space complexity. This observation is especially relevant in pangenomics, where the datasets often do not fit in the computer memory.

Here, we show another crucial advantage of prefix-free graphs. Although they do not offer any improvement of theoretical guarantees in the worst case, in practice, they often represent the pangenome in a substantially smaller space and allow us to generate the suffix array values one by one, possibly using the values directly in subsequential computation or storing them in compressed form. This iteration can be done without ever expanding the pangenome to its full textual representation in space proportional to the sum of segment lengths and the sum of path lengths.

```
segment IDs: 00000111122222233334444445555556
segment join: ACAC#ACG#ACT..#CAC#CGAC#CGTAC#$
segment positions: 01234012301234501230123401234 50
```

Figure 3: Segment join of segments from the running example, corresponding segment IDs, and positions. Permuting IDs and positions according to the `ISA` results in the segment ID array (`ID`) and the segment position array (`pos`) shown in Table 1.

3.1. Iterator preparation

To prepare the iterator of a suffix array of the pangenome from a prefix-free graph, we need to create several data structures. First, we concatenate all the segments into a single string using a separator `#` and append a sentinel `$`. We will call this concatenation *segment join*. An example of a segment join is in Figure 3.

Next, we calculate the segment join's suffix array and the longest common prefix array [26, 27]. For both of these arrays, there exist algorithms with linear time and space complexity which we can use. We note that these linear complexities are proportional to the length of the segment join, which is usually much smaller than the original pangenome.

Next, for each suffix of the segment join, we need to calculate the corresponding segment ID and position values. The value segment ID represents in what segment the current suffix starts, and the value segment position represents at what position in that particular segment the current suffix starts. These arrays can be computed using an inverse permutation of a suffix array `ISA` (Equation 1) of a segment join in linear time.

$$\texttt{ISA}[\texttt{SA}[i]] = i \tag{1}$$

To illustrate the procedure, consider the segment join of our running example from Figure 3. Each position of the join can be assigned a segment ID and a position in the current segment by linearly scanning the segment join and incrementing the ID and position accordingly. Then, applying the ISA to these arrays changes the order of computed values in correspondence to the sorted suffixes. The resulting suffix array (`SA`), longest common prefix array (`LCP`), segment ID array (`ID`) and segment positions array (`pos`) are stored in a *suffix table* as shown in Table 1.

In the suffix table, one row can represent multiple positions of the pangenome. To identify these positions, we store some additional information in a *segment table*. For each segment of the prefix-free graph, we store its length, starting positions in the pangenome and ranks of the right contexts of

Table 1
Suffix table with the suffix array (SA), longest common prefix array (LCP), segment ID array (ID) and segment positions array (pos) of the running example. Suffixes of segment join from position SA[i] added for easier interpretation. Darker colours represent the suffix of the current segment.

i	SA[i]	LCP[i]	ID[i]	pos[i]	segment join[SA[i]..]
0	30	-1	6	0	$
1	29	0	5	5	#$
2	4	1	0	4	#ACG#ACT..#CAC#CGAC#CGTAC#$
3	8	3	1	3	#ACT..#CAC#CGAC#CGTAC#$
4	14	1	2	5	#CAC#CGAC#CGTAC#$
5	18	2	3	3	#CGAC#CGTAC#$
6	23	3	4	4	#CGTAC#$
7	13	0	2	4	.#CAC#CGAC#CGTAC#$
8	12	1	2	3	..#CAC#CGAC#CGTAC#$
9	27	0	5	3	AC#$
10	2	3	0	2	AC#ACG#ACT..#CAC#CGAC#CGTAC#$
11	16	3	3	1	AC#CGAC#CGTAC#$
12	21	5	4	2	AC#CGTAC#$
13	0	2	0	0	ACAC#ACG#ACT..#CAC#CGAC#CGTAC#$
14	5	2	1	0	ACG#ACT..#CAC#CGAC#CGTAC#$
15	9	2	2	0	ACT..#CAC#CGAC#CGTAC#$
16	28	0	5	4	C#$
17	3	2	0	3	C#ACG#ACT..#CAC#CGAC#CGTAC#$
18	17	2	3	2	C#CGAC#CGTAC#$
19	22	4	4	3	C#CGTAC#$
20	1	1	0	1	CAC#ACG#ACT..#CAC#CGAC#CGTAC#$
21	15	4	3	0	CAC#CGAC#CGTAC#$
22	6	1	1	1	CG#ACT..#CAC#CGAC#CGTAC#$
23	19	2	4	0	CGAC#CGTAC#$
24	24	2	5	0	CGTAC#$
25	10	1	2	1	CT..#CAC#CGAC#CGTAC#$
26	7	0	1	2	G#ACT..#CAC#CGAC#CGTAC#$
27	20	1	4	1	GAC#CGTAC#$
28	25	1	5	1	GTAC#$
29	11	0	2	2	T..#CAC#CGAC#CGTAC#$
30	26	1	5	2	TAC#$

Table 2
Segment table containing an ID, length, starting positions in the pangenome and ranks of the right contexts of these positions for each segment.

id	length	starts	right context ranks
0	4	9	9
1	3	15, 1	13, 14
2	5	18, 5, 11	1, 2, 3
3	3	8, 14, 0	4, 5, 6
4	4	16	7
5	5	2	8

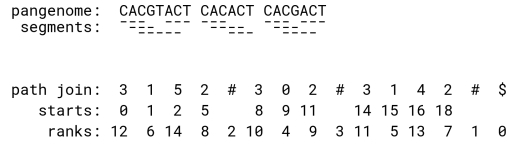

Figure 4: Path join of paths from the running example and the corresponding starts and ranks. The concatenated pangenome and segments are shown above to clarify the meaning of the start values.

these positions. To calculate the starting positions and the ranks of the right contexts, we use a *path join*. Similarly to segment join, a path join is a concatenation with delimiters # and sentinel $, but now constructed by concatenating the paths. An example of a path join for our running example is in Figure 4. Then, starting positions can be calculated by cumulatively summing the lengths of the segments in path join and subtracting the overlaps.

The computation of ranks is more involved. It uses the normalized form of prefix-free graphs since

it relies on a lexicographically smaller ID in a path representing a lexicographically smaller segment. We construct the suffix array of the path join and find its inverse permutation `ISA`. `ISA` gives us ranks for each position in the path join. To determine the rank of the right context for position i, we take the value of $\texttt{ISA}[i+1]$. Finally, we store the starts and ranks sorted by the rank values in the segment table as shown in Table 2.

3.2. Iteration

With the previous tables stored in memory, we have all the necessary ingredients to generate the suffix array value by value.

Each row in the suffix table represents a suffix of a particular segment. There are four cases of what the first position of these suffixes can represent within the segments:

- the sentinel `$`
- a separator `#`
- a position within the last k characters of a segment
- a position outside the last k characters, separator and sentinel

Since the pangenome has no corresponding position for the sentinel or separator characters, we can skip the first rows representing them.

In the third case, the position is inside the trigger word or the sentinels appended during the graph construction. The positions inside the trigger words are represented twice in the suffix table, once at the end of a segment and a second time at the beginning of the following segment in the pangenome. These ending positions can violate the prefix-free property of the segment suffixes and, therefore, can be sorted incorrectly. Skipping through these positions ensures the prefix-free property for the rest of the suffixes and also avoids double reporting. Therefore, if the length of a current segment suffix is smaller or equal to the size of the trigger words k, we skip the row as in the previous cases. This choice also plays nicely with the previous choice of appending k sentinels during the construction of a prefix-free graph, as these positions will not get reported either.

Finally, in the last case, we report the suffix array values. The suffix table can be partitioned into blocks of the same segment suffixes. For example, consider rows 20 and 21, which form a single block. All other blocks in the running example consist of single rows; therefore, we call them singletons.

This partitioning leads to three cases:

- a singleton block with segment suffix occurring only once in the whole pangenome
- a singleton block with segment suffix occurring several times in the pangenome
- a non-singleton block

In the first case, we must report only a single suffix array value. Given the row index i, this value can be calculated with Equation 2.

$$\texttt{SA value} = \texttt{starts}[\texttt{ID}[i]] + \texttt{pos}[i] \tag{2}$$

As an example, consider the row 13 in Table 1, the first row yielding a SA value. Its segment ID is 0, and from Table 2, we see only one occurrence of segment 0 with starting position 9 in the pangenome. The offset from the start of a segment $\texttt{pos}[13]$ is 0. Summing these two values, we get the first value of a suffix array $9 + 0 = 9$ corresponding to the lexicographically smallest suffix of a pangenome $P[9..] = \texttt{ACACT}$.

The second, slightly more complex case is a singleton block representing a segment suffix with several occurrences in the pangenome. In this case, we must report as many suffix array values as the number of occurrences. Because the starting segment positions in the segment table are sorted based on their right context rank, we can iterate through these starting positions and apply Equation 2 to each of them.

As an example, consider the row 14 in Table 1. This suffix occurs twice in the pangenome in segments starting at positions 15 and 1. Since the offset from the start of a segment $\texttt{pos}[14]$ is 0, we report a suffix array values $15+0 = 15$ and $1+0 = 1$, corresponding to the suffixes $P[15..] = \texttt{ACGACT}$ and $P[1..] = \texttt{ACGTACT}$.

In the last case, we have a non-singleton block representing suffixes of several segments, possibly with multiple occurrences. These suffixes represent identical substrings in the pangenome. Here, we report a suffix array value for each of the substrings. To identify the first value, we must find the starting position with the smallest right context rank. Because the ranks are sorted, this procedure is similar to the merging phase of a merge sort. Therefore, to iterate through all suffix array values in the block, we always identify the segment start with the next smallest right context rank and apply Equation 2 to this segment start.

As an example, consider the block of rows [20..21] in Table 1. The relevant segment IDs are 0 and 3, with segment starts at positions 9, 8, 14 and 0. The right context ranks from smallest to highest are $4, 5, 6, 9$ with the corresponding segment starts $8, 14, 0, 9$. Applying Equation 2 to these segment starts yields a suffix array values $8 + 0 = 8$, $14 + 0 = 14$, $0 + 0 = 0$ and $9 + 1 = 10$, represent-

#seqs	pggb	vg	minigraph-cactus	pfg
4	0m 12s	0m 1s	3m 47s	0m 1s
16	0m 19s	0m 1s	6m 30s	0m 1s
64	1m 20s	0m 2s	13m 30s	0m 1s
256	7m 20s	0m 55s	41m 49s	0m 3s

Table 3
Comparison of running times of pangenomic graph construction tools. Wall clock times are measured by the built-in bash command `time` and rounded up to the nearest second. In addition to the measured times, pggb and vg need preprocessing of the pangenome. pggb requires the fasta file to be indexed. vg builds a pangenome from a VCF file or a multiple sequence alignment.

ing pangenome suffixes $P[8..] = \texttt{CACACT}$, $P[14..] = \texttt{CACGACT}$, $P[0..] = \texttt{CACGTACT}$, and $P[10..] = \texttt{CACT}$.

4. Results

We implemented the prefix-free graphs as a Rust package. This package contains two binary crates and one library crate. Binary crates are executables that serve the purpose of creating prefix-free graphs from FASTA and GFA formats. The binary crates are named `fasta2pfg` and `gfa2pfg`, and their usage is as follows:

```
fasta2pfg -t triggers.txt < pangenome.fna >
↪  pfg.gfa

gfa2pfg -t triggers.txt < pangenome.gfa >
↪  pfg.gfa
```

We compared the running time of the construction algorithm with several pangenome construction tools, namely with the PanGenome Graph Builder [9], VG [8], and Minigraph-Cactus [11]. For comparison, we used up to 256 SARS-CoV-2 sequences. These sequences have lengths of around 30 kbp and high nucleotide similarity. We used stop codons (`TAA`, `TAG`, and `TGA`) as trigger words to construct prefix-free graphs. The resulting running times are shown in Table 3.

The library crate provides an interface for working with prefix-free graphs, mainly an iterator of a suffix array in sublinear space and linear time. The library can be used from within the Rust programming language as follows:

```
let pfg = PFG::load("pfg.gfa");

for (i, (sa_i, id_i, pos_i)) in
↪  pfg.iter().enumerate() {
    println!("{}\t{}\t{}\t{}", i, sa_i,
    ↪  id_i, pos_i);
}
```

In addition to the suffix array value, the iterator provides the segment ID and position values. These are useful for the computation of several related data structures. Similarly to Boucher et al. [16], we can use them to compute the Burrows-Wheeler transform $\texttt{BWT}[i]$ by storing the preceding characters of each segment and then reporting the character at position $\texttt{pos}[i]-1$ of a segment $\texttt{id}[i]$. This computation allows for space-efficient construction of r-index [28] similar to MONI implementation [17], space-efficient construction of Wheeler graphs similar in nature to the implementation in Goga and Baláž [13], and with the use of predecessor queries, space-efficient construction of a tag array introduced in Goga et al. [18].

5. Conclusion

An increasing abundance of available genomic data leads to the need for new data structures to utilize all the contained information to its full potential. Pangenome graphs have been proposed as a solution capable of representing these datasets, exploiting the repetitiveness of a collection of related genomes for efficient storage while preserving and sometimes even highlighting the underlying variations. However, this shift in the bioinformatics field from linear sequences to pangenomic graphs brings challenges.

The enormous sizes of pangenomes make the construction of pangenomic graphs nontrivial and often one of the major bottlenecks in the analysis. Current tools rely on well-known computational steps, such as all-against-all alignment [9], multiple sequence alignment [11], or variant calling [8]. All these steps are computationally expensive and do not scale well with rapidly growing datasets. Furthermore, variant calling relies on a linear reference, which can introduce a reference bias.

Here, we introduced the prefix-free graph as a standalone data structure and showed how to build it from currently available pangenomic datasets. In comparison to the other tools, it offers several

crucial advantages. The construction of prefix-free graphs does not require any alignment or variant calling. Instead, it avoids these expensive steps by employing a set of trigger words, which split the sequences according to their contexts. The time complexity of this construction is linear with respect to the input size, and the space complexity is sublinear, proportional only to the size of the resulting data structure.

The novelty of prefix-free graphs brings several possible directions for future research, with the two main directions being the connection of prefix-free graphs to stringology and the choice of trigger words. In Chapter 3, we showed how to use prefix-free graphs to build a suffix array of a pangenome in sublinear space. Suffix arrays are at the core of many efficient and well-established string algorithms in bioinformatics, such as read mapping and pattern matching. We expect this connection will facilitate the development of similar algorithms on pangenomic datasets, supporting the paradigm shift. This direction will require exploring which additional string algorithms can use prefix-free graphs to improve their applicability to these vast and repetitive datasets. An exciting attempt may be to map reads to prefix-free graphs in a similar fashion as the popular tool BWA [29] maps reads to a linear reference.

The second direction, the choice of trigger words, is, so far, mostly undiscovered territory. Trigger words offer great flexibility in the construction process, so a better understanding of their selection is crucial for prefix-free graphs. From the computational perspective, a set of trigger words minimizing the size of the resulting graph is desirable since it would allow the analysis of larger datasets. On the other hand, the construction flexibility could be used to create prefix-free graphs based on biologically significant strings. In our experiments, we used stop codon sequences as the trigger words. However, experiments with other motives, such as different binding sites, recombination hotspots, or repetitive elements, could illuminate the possibility of capturing particular biological phenomena with prefix-free graphs. Moreover, integrating the strand information in the graph construction may be beneficial for further reducing the graph size and capturing biological phenomena such as inversion. This integration could be achieved by considering both the forward and reverse complement of the trigger words.

We believe the characteristics of prefix-free graphs are highly valueable and will have a significant impact when dealing with massive datasets, moving us closer to the ultimate goal of computational pangenomics.

6. Online Resources

The sources for the prefix-free graphs are available via

- https://github.com/andynet/pfg.

Acknowledgments

This research was funded by a grant from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956229 (ALPACA) and by a grant from Slovak Research Grant Agency VEGA 1/0538/22.

References

[1] H. Tettelin, V. Masignani, M. J. Cieslewicz, C. Donati, D. Medini, N. L. Ward, S. V. Angiuoli, J. Crabtree, A. L. Jones, A. S. Durkin, et al., Genome analysis of multiple pathogenic isolates of streptococcus agalactiae: implications for the microbial "pan-genome", Proceedings of the National Academy of Sciences 102 (2005) 13950–13955.

[2] B. T. Lau, D. Pavlichin, A. C. Hooker, A. Almeda, G. Shin, J. Chen, M. K. Sahoo, C. H. Huang, B. A. Pinsky, H. J. Lee, et al., Profiling sars-cov-2 mutation fingerprints that range from the viral pangenome to individual infection quasispecies, Genome medicine 13 (2021) 1–23.

[3] B. E. Dutilh, C. C. Thompson, A. C. Vicente, M. A. Marin, C. Lee, G. G. Silva, R. Schmieder, B. G. Andrade, L. Chimetto, D. Cuevas, et al., Comparative genomics of 274 vibrio cholerae genomes reveals mobile functions structuring three niche dimensions, BMC genomics 15 (2014) 1–11.

[4] M. F. Danilevicz, C. G. T. Fernandez, J. I. Marsh, P. E. Bayer, D. Edwards, Plant pangenomics: approaches, applications and advancements, Current opinion in plant biology 54 (2020) 18–25.

[5] T. Wang, L. Antonacci-Fulton, K. Howe, H. A. Lawson, J. K. Lucas, A. M. Phillippy, A. B. Popejoy, M. Asri, C. Carson, M. J. Chaisson, et al., The human pangenome project: a global resource to map genomic diversity, Nature 604 (2022) 437–446.

[6] Computational pan-genomics: status, promises and challenges, Briefings in bioinformatics 19 (2018) 118–135.

[7] D. M. Church, V. A. Schneider, K. M. Steinberg, M. C. Schatz, A. R. Quinlan, C.-S. Chin, P. A. Kitts, B. Aken, G. T. Marth, M. M. Hoffman, et al., Extending reference assembly models, Genome biology 16 (2015) 1–5.

[8] E. Garrison, J. Sirén, A. M. Novak, G. Hickey, J. M. Eizenga, E. T. Dawson, W. Jones, S. Garg, C. Markello, M. F. Lin, et al., Variation graph toolkit improves read mapping by representing genetic variation in the reference, Nature biotechnology 36 (2018) 875–879.

[9] E. Garrison, A. Guarracino, S. Heumos, F. Villani, Z. Bao, L. Tattini, J. Hagmann, S. Vorbrugg, S. Marco-Sola, C. Kubica, et al., Building pangenome graphs, bioRxiv (2023) 2023–04.

[10] B. Paten, M. Diekhans, D. Earl, J. S. John, J. Ma, B. Suh, D. Haussler, Cactus graphs for genome comparisons, Journal of Computational Biology 18 (2011) 469–481.

[11] G. Hickey, J. Monlong, J. Ebler, A. M. Novak, J. M. Eizenga, Y. Gao, T. Marschall, H. Li, B. Paten, Pangenome graph construction from genome alignments with minigraph-cactus, Nature Biotechnology (2023) 1–11.

[12] T. Gagie, G. Manzini, J. Sirén, Wheeler graphs: A framework for bwt-based data structures, Theoretical computer science 698 (2017) 67–78.

[13] A. Goga, A. Baláž, Prefix-free parsing for building large tunnelled wheeler graphs, in: 22nd International Workshop on Algorithms in Bioinformatics (WABI 2022), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[14] M. Burrows, A block-sorting lossless data compression algorithm, SRC Research Report, 124 (1994).

[15] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proceedings 41st annual symposium on foundations of computer science, IEEE, 2000, pp. 390–398.

[16] C. Boucher, T. Gagie, A. Kuhnle, B. Langmead, G. Manzini, T. Mun, Prefix-free parsing for building big bwts, Algorithms for Molecular Biology 14 (2019) 1–15.

[17] M. Rossi, M. Oliva, B. Langmead, T. Gagie, C. Boucher, Moni: A pangenomic index for finding maximal exact matches, Journal of Computational Biology 29 (2022) 169–187.

[18] A. Goga, A. Baláž, A. Petescia, T. Gagie, Maria: Multiple-alignment r-index with aggregation, arXiv preprint arXiv:2209.09218 (2022).

[19] J. Kornblum, Identifying almost identical files using context triggered piecewise hashing, Digital investigation 3 (2006) 91–97.

[20] A. V. Aho, M. J. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (1975) 333–340.

[21] J. Kärkkäinen, P. Sanders, Simple linear work suffix array construction, in: Automata, Languages and Programming: 30th International Colloquium, ICALP 2003 Eindhoven, The Netherlands, June 30–July 4, 2003 Proceedings 30, Springer, 2003, pp. 943–955.

[22] G. Nong, S. Zhang, W. H. Chan, Linear suffix array construction by almost pure induced-sorting, in: 2009 data compression conference, IEEE, 2009, pp. 193–202.

[23] S. J. Puglisi, W. F. Smyth, A. H. Turpin, A taxonomy of suffix array construction algorithms, acm Computing Surveys (CSUR) 39 (2007) 4–es.

[24] F. A. Louza, S. Gog, G. P. Telles, Inducing enhanced suffix arrays for string collections, Theoretical Computer Science 678 (2017) 22–39.

[25] Y. Mori, libdivsufsort, 2008. URL: https://github.com/y-256/libdivsufsort, accessed on 2023-06-23.

[26] G. M. Landau, T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park, Linear-time longest-common-prefix computation in suffix arrays and its applications, in: Combinatorial Pattern Matching: 12th Annual Symposium, CPM 2001 Jerusalem, Israel, July 1–4, 2001 Proceedings 12, Springer, 2001, pp. 181–192.

[27] G. Manzini, Two space saving tricks for linear time lcp array computation, in: Scandinavian Workshop on Algorithm Theory, Springer, 2004, pp. 372–383.

[28] T. Gagie, G. Navarro, N. Prezza, Fully functional suffix trees and optimal text searching in bwt-runs bounded space, Journal of the ACM (JACM) 67 (2020) 1–54.

[29] H. Li, R. Durbin, Fast and accurate short read alignment with burrows–wheeler transform, bioinformatics 25 (2009) 1754–1760.